From: AAAI Technical Report WS-02-05. Compilation copyright © 2002, AAAI (www.aaai.org). All rights reserved.

Diagnostics with answer sets: Dealing with unobservable fluents

Michael Gelfond and Richard Watson
Dept. of Computer Science
Texas Tech University
Lubbock, TX 79409, USA
{mgelfond, rwatson}@cs.ttu.edu

Abstract

In this paper we present the architecture for an agent capable of operating a physical device and performing diagnosis and repair of the device when malfunctions occur. We expand the algorithm for diagnostics and repair from our previous work to domains in which the states of some components are not observable. The algorithm, based on our new formalism of testing, employs the multiple computation of stable models.

Introduction

Recently, answer set programming (Marek & Truszczynski 1999; Niemelä 1999) was applied to the development of an architecture for a software agent (Baral & Gelfond 2000) capable of performing diagnosis and repair on a physical device it operates (Balduccini, Galloway, & Gelfond 2001; Balduccini & Gelfond 2002). In this approach the agent's knowledge about the world and its own abilities and goals and recorded by a program of A-Prolog - a language of logic programs under the answer set (stable model) semantics (Gelfond & Lifschitz 1988; 1991). Various tasks of the agent (detecting inconsistencies, finding candidate diagnoses, generating tests, planning for goals, etc.) can all be reduced to finding stable models of the corresponding logic programs. In recent years, systems to compute stable models, such as SMODELS, DLV, and DeReS (Niemelä & Simons 1997; Citrigno *et al.* 1997; Cholewinski, Marek, & Truszczyński 1996) have progressed to a point where they are usable for relatively large problems which allowed application of answer set programming to a wide variety of problems.

The chief purpose of this paper is to expand the previous work on diagnostic agents to allow diagnosis, testing, and repair in the situation when not all fluents[1] in the domain are directly observable. We define a new notion of a test and present algorithms for diagnostic reasoning with testing. The algorithms are similar to those in (Balduccini & Gelfond 2002) - the agent's tasks are reduced to finding answer sets of logic programs. As in these previous works, knowledge about the agents domain will be represented using an action language (Gelfond & Lifschitz 1998) and automatically translated into A-Prolog. It should be noted that this work is ideologically similar, yet technically different, from other recent results (Baral, McIlraith, & Son 2000; McIlraith 1997; 1998; McIlraith & Scherl 2000; Otero & Otero 2000; Thielscher 1997) addressing similar problems. As in our previous work, we assume that agent and environment satisfy the following conditions:

1. The agent's environment can be viewed as a transition diagram whose states are sets of fluents and whose arcs are labeled by actions.
2. The agent is capable of making correct observations, performing actions, and remembering the domain history.

These assumptions determine the structure of the agent's knowledge base.

The first part of the knowledge base, called an *action (or system) description*, specifies the transition diagram of the system. It contains descriptions of the domain's actions and fluents, together with the definition of possible successor states to which the system can move after the execution of an action from a given state.

The second part of the agent's knowledge, called the *recorded history*, contains the record of actions performed by the agent together with the agent's observations. This defines a collection of paths within the transition diagram which can be interpreted as the system's possible pasts. If the agent has complete knowledge of his environment and the domain's action are deterministic then there is only one such path.

The final part of the agent's knowledge base contains the agent's goals.

Using and updating this knowledge, the agent operates by repeatedly executing the following steps:

1. observe the world and interpret the observations;
2. select a goal;
3. plan;
4. execute part of the plan.

It is within the first step of this loop, when the observations contradict the agent's belief about the state of the system, that the agent must perform diagnosis. To illustrate principles presented in the paper we will use the following example.

Copyright © 2002, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

[1] Fluents are properties of the domain whose truth value may depend on time.

The Circuit Example

Consider a simple circuit which contains a battery, a switch, a resistor, and a light bulb (see figure 1). When the switch is closed, the light will turn on (unless there is a problem within the system). The agent is capable of opening and closing the switch and replacing the light bulb and resistor with different ones. The three possible exogenous actions[2] in the domain are: the bulb burning out, the resistor shorting, or the resistor breaking. The resistor can not be both broken and shorted at the same time, and having the switch closed when the resistor is shorted will cause the bulb to burn out. Assume that the agent knows that initially all of the components are in good working order and that the switch is open. Suppose the agent then closes the switch but the light does not come on. This contradicts the agents expectation about what should have happened, hence some exogenous action(s) must have happened in the system that would explain the behavior.

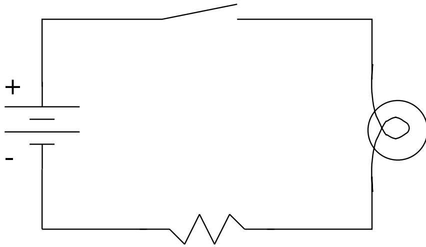

Figure 1: The Circuit Example

The goal of this paper is to develop methods for modeling the agent's behavior after such a discovery, especially in situations where the agent cannot directly observe the condition of some of the components. We will start by presenting a mathematical model of an agent and its environment. In what follows we assume some knowledge of action languages (Gelfond & Lifschitz 1998).

Modeling the domain

This section gives a short overview of diagnostic domains ((Balduccini & Gelfond 2002)). We limit ourselves to *non-intrusive* domains in which the *agent's environment does not normally interfere with his work* and *the agent normally observes all of the domain occurrences of exogenous actions*. The agent is, however, aware of the fact that these assumptions can be contradicted by observations. As a result the agent is ready to observe, test, and take into account occasional occurrences of exogenous 'breaking' actions. Moreover, discrepancies between expectations and observations may force him to conclude that some exogenous actions in the past remained unobserved.

By a *domain signature* we mean a tuple

$$\Sigma = \langle C, F, \mathcal{O}, \mathcal{I}, \mathcal{F}, \mathcal{A}_s, \mathcal{A}_e \rangle$$

of finite sets where $\mathcal{O}, \mathcal{I}$ are subsets of $\mathcal{F}$ and all other sets are disjoint. Elements of C are called device *components* and used to name various parts of the device; F contains possible *faults* of these components. Elements of $\mathcal{A} = \mathcal{A}_s \cup \mathcal{A}_e$ are called *elementary actions*; $\mathcal{A}_s$ contains those which are executable by the agent while actions from $\mathcal{A}_e$ are exogenous. Subsets of A are referred to as *actions*. Intuitively, execution of an action a corresponds to the simultaneous execution of its components. (When convenient, an elementary action a will be identified with $\{a\}$). Elements of $\mathcal{F}$ are referred to as *fluents*. They denote dynamic properties of the domain. We assume that $\mathcal{F}$ contains special fluents $\perp$ (which stands for falsity), *observable*(f) which say that the value of a fluent $f \in \mathcal{O}$ can be actually observed, *executable*(a) which states when an elementary action a can actually be performed, and $ab(c, f)$ which says that the device's component c has a fault f. (The use of ab in diagnosis goes back to (Reiter 1987)). Elements of $\mathcal{O}$ are called *observable* fluents - their values can, under certain conditions, be observed by the agent. We assume that *observable*(f) and *executable*(a) are in $\mathcal{O}$. Fluents from $\mathcal{I}$ are called *inertial* - they are subject to the Inertia Axioms of (McCarthy & Hayes 1969).

By *fluent literals* we mean fluents and their negations (denoted by $\neg f$). The set of literals formed from a set $X \subseteq \mathcal{F}$ will be denoted by $lit(X)$. A set $Y \subseteq lit(\mathcal{F})$ is called *complete* if for any $f \in \mathcal{F}$, $f \in Y$ or $\neg f \in Y$; Y is called *consistent* if $\perp \notin Y$ and there is no f such that $f, \neg f \in Y$.

The possible states of the agent's domain and changes caused by different actions are modeled by a transition diagram over signature Σ - a directed graph T such that:

1. the states of T are labeled by complete and consistent sets of fluent literals from Σ corresponding to possible physical states of the domain.
2. the arcs of T are labeled by actions from Σ.

Paths of a transition diagram T correspond to *possible trajectories* of the domain. Such paths have the form $\sigma_0, a_0, \sigma_1, \ldots, a_{n-1}, \sigma_n$, where each σ_i is state of T and each a_i is an action. A particular trajectory,

$$W = \langle \sigma_0^W, a_0^W, \sigma_1^W, \ldots, a_{n-1}^W, \sigma_n^W, \ldots \rangle,$$

called the *actual trajectory*, corresponds to the actual behavior of the domain.

Definition 1 (Diagnostic Domain) By a *diagnostic domain* we mean a triple $\langle \Sigma, T, W \rangle$ where Σ is a domain signature, T is a transition diagram over Σ, and W is the domain's actual trajectory.

To design an intelligent agent associated with a diagnostic domain $S = \langle \Sigma, T, W \rangle$ we need to supply the agent with the knowledge of Σ, T, and the recorded history of S up to a current point n. Elements of Σ can normally be defined by a simple logic program. Finding a concise and convenient way to define the transition diagram of the domain is somewhat more difficult. While our approach is applicable to a variety of action languages, we will use a variant of action language $\mathcal{B}$ from (Gelfond & Lifschitz 1998).

[2] By exogenous actions we mean action which are not performed by the agent. Such actions may either be natural actions or those performed by other agents operating in the same domain.

Describing the diagram

System descriptions of our language are sets of statements of the form:

1. *causes*$(a, l_0, [l_1, \dots, l_n])$,
2. *caused*$(l_0, [l_1, \dots, l_n])$,
3. *definition*$(l_0, [l_1, \dots, l_n])$

where a is an elementary action and each l_i is a fluent literal. A literal l_0 is called the *head* and the set $\{l_1, \dots, l_n\}$ is called the *body* of the corresponding statement. The first statement, called a *dynamic causal law*, states that 'if action a is performed in a state in which preconditions $l_1, \dots, l_n$ are true, then l_0 will be true in the resulting state.' If $l_0 = \bot$ this law is referred to as *executability precondition*. and sometimes written as

impossible_if$(a, [l_1, \dots, l_n])$.

The second statement, called a *static causal law (or state constraint)*, states that 'if $l_1, \dots, l_n$ are true in a state σ, then l_0 must also be true in σ.' If $l_0 = \bot$ the law states that 'there is no state containing $l_1, \dots, l_n$'.

The third sentence is called a *definition proposition*. The set $def(l_0)$ of definition propositions with the head l_0 can be viewed as an explicit definition of l_0 - l_0 is true in a state σ iff the body of at least one proposition from $def(l_0)$ is true in σ. We will assume that the heads of definition propositions are formed by non-inertial fluents. (Under this assumption such propositions can be replaced by an (often large) collection of static causal laws (Watson 1999).)

The causal laws of a system description SD can be divided into two parts. The first part, SD_n, contains laws describing normal behavior of the system. Their bodies usually contain fluent literals of the form $\neg ab(c, f)$. The second part, SD_b, describes effects of exogenous actions. Such laws normally contain relation ab in the head or positive parts of the bodies. (To simplify our further discussion we only consider exogenous actions capable of causing malfunctioning of the system's components. The restriction is however inessential and can easily be lifted.)

Notice that, to correctly capture the meaning of the fluent *executable*(a), the system description must be written so that it ensures that $\neg$*executable*(a) is true in a state σ iff there is an executability precondition *impossible_if*$(a, [l_1, \dots, l_n])$ such that $l_1, \dots, l_n$ hold in σ. This can be achieved by assuming that every system description SD, of our language satisfy the following condition: for every executability precondition *impossible_if*$(a, [l_1, \dots, l_n]) \in SD$, SD also contains a proposition *definition*$(\neg$*executable*$(a), [l_1, \dots, l_n])$. In our further examples these propositions will not be explicitly written.

Due to the space limitation we will not give the precise definition of the transition diagram defined by a system description SD. The semantics is very close to that of language $\mathcal{B}$ of (Gelfond & Lifschitz 1998). For the precise meaning of definition propositions one is referred to (Watson 1999).

Sample System Description

As an example consider a (simplified) system description SD of the system from Figure 1 Components:

$b_1, b_2, \dots$ (a collection of light bulbs)
$r_1, r_2, \dots$ (a collection of resistors)

In what follows C's will stand for components, B's for bulbs, R's for resistors, and F for an arbitrary fluent. Although there is also a switch and battery in the circuit, for this simple example we will not name them.

Fluents:
closed - the switch is closed
in_circ(C) - a component C is in the circuit
ab$(B, burnt)$ - a light bulb, B, is burned out
ab$(R, shorted)$ - a resistor, R, is shorted
ab$(R, broken)$ - a resistor, R, is broken

Agent Actions:
replace(C_1, C_2) - replaces component C_1 by a spare component C_2
open - opens the switch
close - closes the switch

Exogenous Actions:
burn - burns out the circuit's bulb
short - shorts out the circuit's resistor
break - breaks the circuit's resistor

Causal laws and executability conditions describing the normal functioning of S:

$$SD_n \left\{ \begin{array}{l} causes(open, \neg closed, [\,]). \\ causes(close, closed, [\,]). \\ causes(replace(C_1, C_2), in_circ(C_2), [\,]). \\ causes(replace(C_1, C_2), \neg in_circ(C_1), [\,]). \\ caused(observable(F), [\,]). \\ definition(lit(B), [closed, in_circ(B), \\ \quad in_circ(R), \neg ab(B, burnt), \\ \quad \neg ab(R, shorted), \neg ab(R, broken)]). \\ impossible_if(open, [\neg closed]). \\ impossible_if(close, [closed]). \\ impossible_if(replace(C_1, C_2), [\neg in_circ(C_1)]). \end{array} \right.$$

Laws describing the effects of exogenous actions:

$$SD_b \left\{ \begin{array}{l} causes(burn, ab(B, burnt), [in_circ(B)]). \\ causes(short, ab(R, shorted), [in_circ(R)]). \\ causes(break, ab(R, broken), [in_circ(R)]). \\ caused(ab(B, burnt), [ab(R, shorted), \\ \quad in_circ(B), in_circ(R)]). \\ caused(\bot, [ab(R, shorted), ab(R, broken)]). \end{array} \right.$$

Later we consider some fluents to be unobservable.

Recording the History

A recorded history, Γ_n, of a system up to step n is a collection of statements of the form:

1. $initially(l)$ - 'fluent literal l is true in the initial state'
2. $obs(l, t)$ - 'observable fluent literal l was observed to be true at moment t';
3. $hpd(a, t)$ - 'elementary action $a \in A$ was observed to have happened at moment t'

where t is an integer in the range $[0, n)$.

A pair, $\langle SD, \Gamma_n \rangle$ is referred to as a *domain description*.

For the following definitions, let S be a diagnostic domain, T be the transition diagram defined by S, $W = \langle \sigma_0^W, a_0^W, \sigma_1^W, \ldots, a_{n-1}^W, \sigma_n^W \rangle$ be the actual trajectory, and Γ_n be a history of S up to moment n.

Definition 2 (Model) A path $\sigma_0, a_0, \sigma_1, \ldots, a_{n-1}, \sigma_n$ in T is a *model* of Γ_n iff

1. if $initially(l) \in \Gamma_n$ then $l \in \sigma_0$
2. if $hpd(a, k) \in \Gamma_n$ then $executable(a) \in \sigma_k$
3. $a_k = \{a : hpd(a, k) \in \Gamma_n\}$;
4. if $obs(l, k) \in \Gamma_n$ then $observable(l) \in \sigma_k$ and $l \in \sigma_k$.

Definition 3 (Consistent) Γ_n is *consistent* (w.r.t. T) if it has a model.

Definition 4 (Sound) Γ_n is *sound* (with respect to S) if, for any literal, l, action, a, and time, t:

1. if $initially(l) \in \Gamma_n$ then $l \in \sigma_0^W$
2. if $obs(l, t) \in \Gamma_n$ then $observable(l) \in \sigma_t^W$ and $l \in \sigma_t^W$, and
3. if $hpd(a, t) \in \Gamma_n$ then $executable(a) \in \sigma_t^W$ and $a \in a_t^W$.

Definition 5 ($\models$ (Entailment)) A fluent literal, l, *holds* in a model, M, at time $t \leq n$ (denoted $M \models h(l, t)$) if $l \in \sigma_t$. Γ_n *entails* $h(l, t)$ (denoted $\Gamma_n \models h(l, k)$) if for every model M of Γ_n, $M \models h(l, k)$.

The definitions are almost identical to that of ((Balduccini & Gelfond 2002)) - the most important difference is the inclusion of $initially(l)$ and the fact that fluents must have been observable at the time they were observed.

With these definitions, we can now look at the history described in the circuit example, examine the models of the domain description, and see what can be entailed.

Sample History Description

For our circuit example, the history, Γ_0, can be encoded by:

$hpd(close), 0)$.
$initially(\neg closed, 0)$.
$initially(\neg lit(B), 0)$.
$initially(in_circ(b_1), 0)$.
$initially(in_circ(r_1), 0)$.
$initially(\neg in_circ(C), 0)$.
$initially(\neg ab(B, burnt), 0)$.
$initially(\neg ab(R, F), 0)$.

where B is any light bulb, R is any resistor, C is any bulb or resistor other than b_1 and r_1, and F is either *shorted* or *broken*.

Given the system description SD of S, and the recorded history, Γ_0, it is easy to see that the path $\langle \sigma_0, close, \sigma_1 \rangle$ is the only model of Γ_1 and that $\Gamma_1 \models h(lit(b_1), 1)$.

Basic Definitions

We now present some definitions needed for diagnosis and testing. The definitions, with minor modifications, are from (Balduccini & Gelfond 2002).

Definition 6 (Configuration) A pair, $\langle \Gamma_n, O_n^m \rangle$, where Γ_n is the recorded history of S up to moment n and O_n^m is a collection of observations made by the agent between n and m, is called a *configuration*.

Definition 7 (Symptom) A configuration, $\mathcal{S} = \langle \Gamma_n, O_n^m \rangle$, is a *symptom* of the system's malfunction if Γ_n is consistent but $\Gamma_n \cup O_n^m$ is not.

Definition 8 (Explanation) An *explanation* (Baral & Gelfond 2000), E, of a symptom, $\mathcal{S}$, is a collection of statements

$$E = \{hpd(a_i, t) : 0 \leq t < n \text{ and } a_i \in A_e\}$$

such that $\Gamma_n \cup O_n^m \cup E$ is consistent.

Definition 9 (Candidate Diagnosis) A *candidate diagnosis* of a symptom, $\mathcal{S}$, is a pair $D = \langle E, \Delta \rangle$ such that E is an explanation of $\mathcal{S}$ and $\Delta = \{(c, f, k) : n \leq k \leq m,\ M \models h(ab(c, f), k)\}$ for some model M of $\Gamma_n \cup O_n^m \cup E$.

Given a candidate diagnosis, D, for convenience we often denote E by act(D) and Δ by faults(D). Moreover, in some cases, we abuse the notation, and identify D with $E \cup \{obs(ab(c, f), k) : (c, f, k) \in \Delta\}$.

Definition 10 (Diagnosis) We say that a candidate diagnosis, $D = \langle E, \Delta \rangle$, is a *diagnosis* of a symptom, $\mathcal{S} = \langle \Gamma_n, O_n^m \rangle$, if all the components in Δ are faulty, i.e., for any $(c, f, k) \in \Delta$, $ab(c, f) \in \sigma_k^W$.

Sample Candidate Diagnosis

Previously we presented a system description, SD, and history, Γ_0, for our circuit example. To continue with this example let us assume that at time point 1 (the time point immediately after the agent closed the switch) the agent observed that the bulb is not lit. Using our notation we have a configuration $\mathcal{S}^0 = \langle \Gamma_0, O_1^1 \rangle$ such that $O_1^1 = \{obs(\neg lit(b_1), 1)\}$. It is easy to see that $\mathcal{S}^0$ is a symptom of a malfunction in the system and that the following is the complete list of candidate diagnoses of $\mathcal{S}^0$:

$D_1 = \langle \{hpd(burn, 0)\}, \{(b_1, burnt)\} \rangle$
$D_2 = \langle \{hpd(short, 0)\}, \{(b_1, burnt), (r_1, shorted)\} \rangle$
$D_3 = \langle \{hpd(break, 0)\}, \{(r_1, broken)\} \rangle$
$D_4 = \langle \{hpd(burn, 0), hpd(short, 0)\}, \{(b_1, burnt), (r_1, shorted)\} \rangle$
$D_5 = \langle \{hpd(burn, 0), hpd(break, 0)\}, \{(b_1, burnt), (r_1, broken)\} \rangle$

At this point, the agent needs to find a diagnosis from the set. This is accomplished through testing.

Testing

When confronted with a number of candidate diagnoses, an agent has to be able to determine what is or is not a diagnosis. In our previous works, (Balduccini, Galloway, & Gelfond 2001; Balduccini & Gelfond 2002) all fluents were assumed to be observable at all times and therefore testing could be done by simply observing the state of the components mentioned in the diagnosis. If a candidate diagnosis predicts that a component should be faulty but the real component is not, then the agent can reject that candidate. The

situation is much more difficult when there are components which are not always observable. An agent can still reject a candidate based on the currently observable fluents, but, if the candidate diagnosis contains any fluents which are not currently observable, there could be insufficient information to accept the candidate as a diagnosis. The agent must find a way to test the suspected components. In what follows we assume that no (relevant) exogenous actions can occur during the testing and repair process.

Definition 11 (Test) A *test* of a candidate diagnosis D of a symptom $\mathcal{S} = \langle \Gamma_n, O_n^m \rangle$ is a pair (α, U) where $\alpha = \{a_1, \ldots, a_k\}$ is a sequence of actions, U is a collection of fluents such that there is a model M of $\mathcal{S} \cup D \cup \{hpd(a_i, m+i : a_i \in \alpha\}$ which entails $h(observable(f), m+k+1)$ for any fluent f from U.

Notice that the simplest case of test is when, for some pair $(c, f, m) \in \Delta$, $ab(c, f)$ is observable in the current state, m. In such a case, $([\ ], \{ab(c, f)\})$ is a test. If this is not the case, then the agent may try to find an α that will lead to a state containing *observable*$(ab(c, f))$. This can be achieved by using a standard planning algorithm (Dimopoulos, Nebel, & Koehler 1997). If α is successfully executed and c is observed to have a fault f the agent may conclude that c was faulty before the test. Of course this conclusion can only be justified assuming that components are not broken by the testing actions. If, contrary to the D based prediction, $ab(c, f)$ is observed to be false, D will be rejected as the candidate diagnosis. If the agent's assumptions are incorrect it can also discover that some action in his test is not executable or that some element of U is not actually observable. This information will be added to the agent's set of observations and will allow the agent to eliminate a number of possible candidate diagnoses of the original symptom, including D. In general *good* tests are supposed to reduce the number of candidate diagnosis of the symptom. An obvious way of eliminating useless tests is to only choose those which are in some way related to elements of Δ. This and other approaches to the problem will be addressed in the full version of the paper.

Diagnostic and Testing Algorithms

In this section we give a brief description of an algorithm for finding a diagnosis of a symptom. We assume a fixed system description SD such that any symptom $\mathcal{S}$ of its malfunctioning has a diagnosis. The algorithm will use two basic routines, *Get_Test*$(\mathcal{S}, D)$ and *Perform_Test*$(\mathcal{S}, T)$. The former finds a test, T, of a diagnosis D of symptom $\mathcal{S}$ and returns nil if no such test is available. The latter attempts to execute actions of T and to observe the resulting fluents. The results of this activity will be recorded in the agent's knowledge base. It is easy to show that the resulting configuration, $\mathcal{S}'$, will remain a symptom. Consider for instance a test $T = \langle a, \{f\} \rangle$. Assuming that f is true after the execution of a, the successful execution of the test will expand S by statements $hpd(a, m)$ and $obs(f, m+1)$. If the execution of a fails, the resulting $\mathcal{S}' = \mathcal{S} \cup obs(\neg executable(a), m)$. Finally, if the agent discovers that f is not observable after the execution of a then $\mathcal{S}' = \mathcal{S} \cup \{hpd(a, m), obs(\neg observable(f, m+1))\}$.

We assume that if a is not executable or f is not observable this fact will always be detected by the agent. Normally, the function *Get_Test* will significantly depend on the agent's knowledge of the domain. For illustrative purposes we describe this function for a domain in which a fault f of a component c can only be observed in a state satisfying some conditions $t(c, f)$ recorded in the agent's knowledge base SD.

First we will need some notation. Recall that in (Balduccini & Gelfond 2002) we defined a logic program, *TEST*(S), which encodes SD together with a configuration $\mathcal{S}$. That paper also contains a proof of correctness of this encoding with respect to the semantics of our action language. We will also need a standard planning module PM (Dimopoulos, Nebel, & Koehler 1997; Lifschitz 1999) (possibly extended by some heuristic information). Finally, by Π we will denote a logic program *Test*$(\mathcal{S} \cup D) \cup goal(t(c, f)) \cup PM$.

function *Get_Test*($\mathcal{S}$: **symptom**,D :**diagnosis**):**diagnosis**;
  begin
    select $\langle c, f \rangle \in$ faults(D);
    if Π is consistent **then**
      select an answer set X of Π;
      return$(a_1, \ldots, a_k : occurs(a_i, m+i) \in X)$;
    else return(nil)
end

The correctness of our encoding, *TEST*$(\mathcal{S})$, together with the correctness of the planning module guarantee correctness of the returned value of *Get_Test*. Slight modifications of the above code will allow us to avoid multiple observances of the same fluents, to limit the length of tests, to restrict tests to actions relevant to $t(c, f)$, etc.

We will also need a routine for finding candidate diagnoses of a symptom. The one presented below is very similar to that in (Balduccini & Gelfond 2002). It uses a logic program $D_0(\mathcal{S})$ =*TEST*$(\mathcal{S}) \cup DM$ where DM is one of the diagnostic modules described in that paper. Recall also that a set of literals X *determines* a candidate diagnosis $\langle E, \Delta \rangle$ if $E = \{hpd(a, t)$:*occurs*$(a, t) \in X, a \in A_e\}$ and $\Delta = \{(c, f, t) : obs(ab(c, f), t) \in X, n \leq t \leq m\}$.

function *Candidate_Diag*($\mathcal{S}$: **symptom** ,
                Q: **set of diagnosis**):**diagnosis**;
{ returns a candidate diagnosis of $\mathcal{S}$ which is not in Q or
  nil if no such candidate diagnosis exists }
var E, X : **history**;
    Δ : **set of faults**;
begin
  if $D_0(\mathcal{S})$ is consistent **then**
    begin
      select an answer set, X, of $D_0(\mathcal{S})$ such that the
        pair $\langle E, \Delta \rangle$ determined by X is not in Q;
      return$(\langle E, \Delta \rangle)$
    end
  else
    return (nil);
end

Correctness of the returned value of the function was proven in (Balduccini & Gelfond 2002).

Next we will define a function *Test* responsible for finding and performing a test of a diagnosis D of symptom $\mathcal{S}$. *Test* may establish that D is a diagnosis or that the appropriatness of D as a diagnosis cannot be discovered by any test. If neither of these outcomes occur it computes a new candidate diagnosis of the (possibly updated) symptom. A more precise description of the behavior of the function is given in Proposition 1 below.

Type : test_outcome = $\{$**y,n,u**$\}$;

```
function Test( var S: symptom,
                  D: diagnosis,
                  var D': diagnosis,
                  var Q: set of diagnosis ):test_outcome;
var T : test;
begin
   if for every candidate diagnosis, D_i of S,
      faults(D) ⊆faults(D_i) then
         return(y);
   T := Get_Test(S, D);
   if T = nil then
      Q := Q ∪ {D};
   else
      Perform_Test(S, T);
   D' :=Candidate_Diag(S, Q);
   if D' = nil then
      return(n)
   else
      return(u);
end
```

Proposition 1 Let $\mathcal{S}_0$ and $\mathcal{S}_1$ be the initial and the final value of parameter $\mathcal{S}$ of *Test*. Then

1. If *Test* returns **y** then D is a diagnosis of $\mathcal{S}_0$.
2. If *Test* returns **n** then the diagnosis of $\mathcal{S}_1$ cannot be tested. (If every symptom of our system has a diagnosis then Q contains a diagnosis of $\mathcal{S}_0$ and $\mathcal{S}_1$.)
3. If *Test* returns **u** then D' is a candidate diagnosis of $\mathcal{S}_1$.

The proof of the proposition will be given in the full paper.

Notice that checking the condition of the first **if** statement of *Test* can be done as follows:

1. compute a stable model of a logic program $\Pi' = D_0(\mathcal{S}) \cup C$ where C is a logic programming constraint

$\{$:- $\{h(ab(c, f), k) : (c, f, k) \in faults(D)\}\}$

2. If a model is found the condition fails. Otherwise it is true.

So, as before, the real computation of the function is reduced to finding stable models of logic programs.

Now we are ready to define the main function of our algorithm which generates and tests the candidate diagnoses of a symptom.

```
function Find_Diag( var S: symptom ) : diagnosis;
{ returns a diagnosis of S or nil if no diagnosis is found.}
var   D, D' : diagnosis;
      Q : set of diagnosis;
      Outcome : test_outcome;
begin
   Q = { };
   D :=Candidate_Diag(S, Q);
   while D ≠ nil
      Outcome:=Test(S, D, D', Q);
      if Outcome= y then
         return(D);
      D := D';
   end {while}
   if D is the only element of Q consistent with S then
      return(D)
   else return(nil)
end
```

It is not difficult to show that, if the function returns D, then it is a diagnosis of both, the original and the final symptoms $\mathcal{S}_0$ and $\mathcal{S}_1$. In the full paper we will give several conditions which will guarantee the function's termination.

Testing in the Circuit Example

To illustrate the algorithm let us consider several versions of our circuit example. The versions will differ only by what fluents are viewed as observable. Let us assume that the agent already discovered a symptom $\mathcal{S}^0$. In our trace we will refer to its diagnoses $D_1, \ldots, D_5$. We will also assume that our function *Get_Test* returns a test with the shortest possible sequence of actions. (Which is common and easy to implement).

We start with a system, S_0, in which all faults are observable. This is a straight forward diagnostic problem and can easily be handled by the previous agent framework of (Balduccini, Galloway, & Gelfond 2001),(Balduccini & Gelfond 2002)). Function *Find_Diag* will simply generate the candidate diagnoses and observe their predicted faults. The process will terminate and return one of $D_1, \ldots, D_5$.

Next we will consider S_1 in which we can observe if bulbs are burnt out but faults of resistors are not observable. Here the previous work is no longer sufficient. In *Find_Diag*, suppose the first candidate diagnosis chosen to check was D_1. D_1 contains only one fault - the light bulb, b_1, being burnt out - and it is observable. The *Test* algorithm will call *Get_Test* which will return $([\ ], \{ab(b_1, burnt)\})$. Suppose that the agent performed the test and discovered that the bulb is not burned out. This information is added to $\mathcal{S}^0$. *Test* then calls *Candidate_Diag* with the new symptom $\mathcal{S}^1$. Notice that, since the bulb was good, the only remaining candidate diagnosis of $\mathcal{S}^1$, D_3, will be saved in D' and returned to *Find_Diag* along with the test outcome **u**. In *Find_Diag*, D then becomes D_3 and *Test* is called again. Since D_3 is the only candidate diagnosis of $\mathcal{S}^1$, it is obviously true that the first **if** statement in *Test* succeeds, **y** is returned, and hence D_3 is returned by *Find_Diag* as a diagnosis of $\mathcal{S}^1$. Although r_1 was not directly observed to be broken, it must be the case (otherwise $\mathcal{S}^1$ would not have a diagnosis). Of course, this is also the only diagnosis of $\mathcal{S}^0$. If the bulb had been observered to be burnt out, the information would be added to $\mathcal{S}^0$ and a candidate diagnosis D' would be chosen and returned

with outcome **u**. If D_1 was chosen again as our D' then the next call to *Test* would return it as a diagnosis. If D_2, D_4, or D_5 was chosen for D', further testing would be performed and the algorithm would continue until a diagnosis was confirmed.

Finally, let us assume that neither faults of bulbs nor resistors are observable. Again let us consider the case when D_1 was the first candidate chosen in *Find_Diag*. As before, *Find_Diag* calls *Test* and *Test* calls *Get_Test*. The system will try to find a test which will help determine if the bulb is bad. One such test would be the pair $([replace(b_1, b_2)], \{lit(b_2)\})$ (replace the bulb in the circuit with one of the good spare bulbs and observe if the new bulb lights up). The information gained while performing the test is added to $\mathcal{S}^0$. Suppose the new bulb did light up. When *Test* calls *Candidate_Diag* to get a candidate, it is easy to see that D_1 is the only candidate which is consistent with $\mathcal{S}^1$. Again, after **u** and D' are returned by *Test*, D will become D_1. It will be returned as a diagnosis on the next call to *Test*. If, in this example, the new light bulb had not lit, D_1 could not be a diagnosis and hence, one of the other candidate diagnoses of $\mathcal{S}^1$ would be chosen as D'. To continue the example informally, suppose D_2 is chosen. We again get a test. Consider the test $(switch(r_1, r_2), lit(b_2))$. If the light bulb is not lit after this action, then r_1 must have been shorted and blew out b_2 as soon as it was put in the socket. Since we know the resistor was shorted, we would also know that b_1 was bad as well. This is due to the fact that no exogenous actions are allowed to happen during the testing. In this case, both D_2 and D_4 are still candidates. Note that it is impossible to differentiate between these two. In both cases the bulb, b_1, is abnormal (burned out) and r_1 is shorted. Since the exogenous actions which caused the faults had no other effect, it is impossible to deduce which set of actions actually happened. Notice however, that they are both, by definition, diagnoses of $\mathcal{S}^2$ (and hence $\mathcal{S}^0$. The one which is chosen by *Candidate_Diag* as the new D' in *Test* will be eventually returned as a diagnosis.

If the bulb had lit when the resistors were switched, then D_2 could not be a diagnosis since r_1 must have been broken, not shorted. The two remaining candidate diagnoses are D_3 and D_5. D_3 is definitely a diagnosis because we know r_1 was broken. If D_5 was chosen next however, we could run the test $([replace(b_2, b_1)], lit(b_1))$. By putting the original bulb back in the circuit whose resistor is now certainly working, we would be sure if b_1 was abnormal or not.

We hope that this discussion facilitated the reader's understanding of the algorithm. In the full paper we hope to present its implementation and show the actual output of the program.

The example also show that the quality of the algorithms depends very substantially on the implementation of *Get_Test*. Assume for instance that, in our example, the light wasn't on because the resistor had shorted and *Get_Test* returns a test which requires changing the bulb. As was discussed above, since the resistor is shorted, not only will the light not be on as a result of this action, but the new bulb will also be blown in the process. We may attempt to avoid this situation by modifying *Get_Test* to insure that performance of a test will not cause new abnormalities if executed in any of the possible current states of the system. Such "safe" tests may, of course, be preferable. But, it can easily be seen that such test do not always exist. We will leave discussion of safe tests and other similar choices and their ramifications for future work.

Conclusion and Future Work

This paper desribes an ongoing work investigating the applicability of answer set programming to the design and implementation of intelligent agents capable of acting in a changing environment. In particular it expands the work on the diagnostic reasoning module of the agent (Balduccini & Gelfond 2002) to allow for repair combined with testing in situations where faults of the system are not always observable. We introduced a new notion of a test which differs somewhat from those available in the literature (McIlraith & Reiter 1992; McIllraith 1994; Baral, McIlraith, & Son 2000; McIlraith & Scherl 2000). We also presented new reasoning algorithm for finding diagnoses of a symptom. The algorithm uses testing and observations to desciminate between various candidate diagnoses. Most of the computational work in the algorithm is accomplished by answer set solvers which compute answer sets of logic programs representing the agent's knowledge. Even though the relationship between our approach and other related work requires further investigation, it is clear that some features of our approach are new. For instance we differ from (Baral, McIlraith, & Son 2000) in our use of action language with fixed ordering of points and parallel actions. In contrast, that paper allows arbitrary interleaving of unobserved actions. While our work seems to be simpler and provide some computational advantages, that of (Baral, McIlraith, & Son 2000) is more general. Our approach also seems to avoid the need for special treatment of knowledge producing actions (as introduced in (McIlraith & Scherl 2000)). In our approach the agent confirms his diagnostics by observations and by establishing that, unless certain faults are present, no explanation of the symptom is possible. The agents from (McIlraith & Scherl 2000) seem to rely more on logical reasoning in sitaution calculus with knowledge producing actions. It will be very interesting to investigate the relationship between these approaches. In our further work we also plan to refine and optimize the algorithm and to investigate their properties. In particular we are interested in finding conditions on the domain and the function *Get_Test* which will guarantee termination and completeness of *Find_Diag*. Other plans include implementation and testing of our diagnostic module.

Acknowledgments

This work was partially supported by United Space Alliance under Contract COC6771311 and Research Grant 26-3502-21.

References

Balduccini, M., and Gelfond, M. 2002. Diagnostic reasoning with a-prolog. submitted for publication.

Balduccini, M.; Galloway, J.; and Gelfond, M. 2001. Diagnosing physical syatems in a-prolog. In *Proceedings of the AAAI Workshop on Answer Set Programming*, 77–83.

Baral, C., and Gelfond, M. 2000. Reasoning agents in dynamic domains. In Minker, J., ed., *Logic-Based Artificial Intelligence*. Kluwer Academic Publishers. 257–279.

Baral, C.; McIlraith, S.; and Son, T. 2000. Formulating diagnostic problem solving using an action language with narratives and sensing. In *Proceedings of the 2000 KR Conference*, 311–322.

Cholewinski, P.; Marek, W.; and Truszczyński, M. 1996. Default reasoning system deres. In *International Conference on Principles of Knowledge Representation and Reasoning*, 518–528. Morgan Kauffman.

Citrigno, S.; Eiter, T.; Faber, W.; Gottlob, G.; Koch, C.; Leone, N.; Mateis, C.; Pfeifer, G.; and Scarcello, F. 1997. The dlv system: Model generator and application frontends. In *Proceedings of the 12th Workshop on Logic Programming*, 128–137.

Dimopoulos, Y.; Nebel, B.; and Koehler, J. 1997. Encoding planning problems in non-monotonic logic programs. In Saraswat, V., and Ueda, K., eds., *Lecture Notes in Artificial Intelligence - Recent Advances in AI Planning, Proceedings of the 4th European Conferenceon Planning, ECP'97*, volume 1348, 169–181.

Gelfond, M., and Lifschitz, V. 1988. The stable model semantics for logic programming. In Kowalski, R., and Bowen, K., eds., *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, 1070–1080.

Gelfond, M., and Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Computing* 365–387.

Gelfond, M., and Lifschitz, V. 1998. Action languages. *Electronic Transactions on Artificail Intelligence* 3(16).

Lifschitz, V. 1999. Action languages, answer sets, and planning. In *The Logic Programming Paradigm: a 25-Year Perspective*. Springer Verlag. 357–373.

Marek, W., and Truszczynski, M. 1999. Stable models and an alternative logic paradigm. In *The Logic Programming Paradigm: a 25-Year Perspective*. Springer Verlag. 375–398.

McCarthy, J., and Hayes, P. 1969. Some philosophical problems from the standpoint of artificial intelligence. In Meltzer, B., and Michie, D., eds., *Machine Intelligence*, volume 4. Edinburgh University Press. 463–502.

McIllraith, S. 1994. Generating tests using abduction. In *Proc. of Kr-94*, 449–460.

McIlraith, S., and Reiter, R. 1992. On tests for hypothetical reasoning. In Hamscher, W. e. a., ed., *Readings in Model-Based Diagnosis*. Morgan Kaufman. 89–96.

McIlraith, S., and Scherl, R. 2000. What sensing tells us: Toward a formal theory of testing in dynamical systems. In *Proceedings of the 17th National Conference on Artificial Intelligence (AAAI '00)*, 483–490.

McIlraith, S. 1997. Representing actions and state constraints in model-based diagnosis. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI '97)*, 43–49.

McIlraith, S. 1998. Explanatory diagnosis conjecturing actions to explain observations. In *Proceedings of the 1998 KR Conference*, 167–177.

Niemelä, I., and Simons, P. 1997. Smodels - an implementation of the stable model and well-founded semantics for normal logic programs. In *Proceedings of the 4th International Conference on Logic Programming and Non-Monotonic Reasoning*, 420–429.

Niemelä, I. 1999. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence* 25(3-4):241–273.

Otero, M., and Otero, R. 2000. Using causality for diagnosis. In *Proceedings of 11th International Workshop on Principles of Diagnosis*, 171–176.

Reiter, R. 1987. A theory of diagnosis from first principles. *Artificial Intelligence* 32(1):57–95.

Thielscher, M. 1997. A theory of dynamic diagnosis. *Linkoping Electronic Articles in Computer and Information Science* 2(11).

Watson, R. 1999. An application of action theory to the space shuttle. In Gupta, G., ed., *Lecture Notes in Computer Science - Proceedings of Practical Aspects of Declarative Languages '99*, volume 1551, 290–304.